Theoretical Computer Science 411 (2010) 3994–4004

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Adapting parallel algorithms to the W-Stream model, with applications to graph problems☆

Camil Demetrescu[a], Bruno Escoffier[b], Gabriel Moruz[c], Andrea Ribichini[a,*]

[a] Dipartimento di Informatica e Sistemistica, Sapienza Università di Roma, Via Ariosto 25, 00185 Rome, Italy
[b] Lamsade, Université Paris Dauphine, Place du Maréchal de Lattre de Tassigny, 75775 Paris Cedex 16, France
[c] Institute for Computer Science, Goethe University, Robert-Mayer-Straße 11-15, 60325 Frankfurt/Main, Germany

ARTICLE INFO

Article history:
Received 27 August 2008
Received in revised form 22 May 2010
Accepted 27 August 2010
Communicated by G. Italiano

Keywords:
Data streaming model
Reductions to parallel algorithms
Graph problems

ABSTRACT

In this paper we show how parallel algorithms can be turned into efficient streaming algorithms for several classical combinatorial problems in the W-Stream model. In this model, at each pass one input stream is read, one output stream is written, and data items have to be processed using limited space; streams are pipelined in such a way that the output stream produced at pass i is given as input stream at pass $i+1$. We first introduce a simulation technique that allows turning efficient PRAM algorithms into optimal W-Stream ones, for many classical combinatorial problems, including list ranking and Euler tour of a tree. For other problems, most notably graph problems, however, this technique leads to suboptimal algorithms. To overcome this difficulty we introduce the *Relaxed* PRAM (RPRAM) computational model, as an intermediate model between PRAM and W-Stream. RPRAM allows every processor to access a non-constant number of memory cells per parallel round, albeit with some restrictions. The RPRAM model, while being more powerful than the PRAM model, can be simulated in W-Stream within the same asymptotic bounds. The extra power provided by RPRAM allows us in many cases to substantially reduce the number of processors, while maintaining the same number of parallel rounds, leading to more efficient W-Stream simulations of parallel algorithms. Our RPRAM technique gives new insights on developing streaming algorithms and yields efficient algorithms for several classical problems in this model including sorting, connectivity, minimum spanning tree, biconnected components, and maximal independent set. In addition to allowing smooth space-passes tradeoffs, our algorithms are also shown, by proving almost-tight communication complexity-based lower bounds in W-Stream, to be optimal up to polylog factors.

1. Introduction

Data stream processing has gained increasing popularity in the last few years as an effective paradigm for processing massive data sets. Huge data streams arise in several modern applications, including database systems, IP traffic analysis, sensor networks, and transaction logs [15,27]. Streaming is an effective paradigm also in scenarios where the input data

☆ Supported in part by the Sixth Framework Programme of the EU under contract number 001907 ('DELIS: Dynamically Evolving, Large Scale Information Systems'), by the Italian MIUR Project 'MAINSTREAM: Algorithms for massive information structures and data streams', by the DFG grant ME 3250/1-1, and by the center for massive data algorithmics (MADALGO), funded by the Danish National Research Foundation. A preliminary version of this paper appears in the Proceedings of the 32nd International Symposium on Mathematical Foundations of Computer Science (MFCS'07).

* Corresponding author.
E-mail addresses: demetres@dis.uniroma1.it (C. Demetrescu), escoffier@lamsade.dauphine.fr (B. Escoffier), gabi@cs.uni-frankfurt.de (G. Moruz), ribichini@dis.uniroma1.it (A. Ribichini).
URLs: http://www.dis.uniroma1.it/~demetres (C. Demetrescu), http://www.lamsade.dauphine.fr/~escoffier (B. Escoffier), http://www.cs.uni-frankfurt.de/~gabi (G. Moruz), http://www.dis.uniroma1.it/~ribichini (A. Ribichini).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.08.030

is not necessarily represented as a data stream. Due to high sequential access rates of modern disks, streaming algorithms can be effectively deployed for processing massive files on secondary storage [17], providing new insights into the solution of computational problems in external memory. In the classical read-only streaming model, algorithms are constrained to access the input data sequentially in one (or few) passes, using only a small amount of working memory, typically much smaller than the input size [17,21,22]. Usual parameters of the model are the working memory size s and the number of passes p that are performed over the data, which are usually functions of the input size. Among the problems that have been studied in this model under the restriction that $p = O(1)$, we recall statistics and data sketching problems (see, e.g., [2,13,14]), which can be typically approximated using polylogarithmic working space, and graph problems (see, e.g., [6,11,12]), most of which require a working space linear in the vertex set size.

Motivated by practical factors, such as availability of large amounts of secondary storage at low cost, a number of authors have recently proposed less restrictive streaming models, where algorithms can both read and write data streams [24]. Among them, we mention the W-Stream model and the StrSort model [1,23]. In the W-Stream model, at each pass we operate with an input stream and an output stream. The streams are pipelined in such a way that the output stream produced at pass i is given as input stream at pass $i + 1$. Despite the use of intermediate streams, which allows achieving effective space-passes tradeoffs for fundamental graph problems (which can be solved with working memory sublinear in the vertex set size, at the expense of increasing the number of passes), most classical lower bounds on the $p \cdot s$ product in read-only streaming hold also in this model [10]. The StrSort model is just W-Stream augmented with a sorting primitive that can be used at each pass to reorder the output stream for free. Sorting provides a lot of computational power, making it possible to solve several graph problems using polylog passes and working space [1]. For a comprehensive survey of algorithmic techniques for processing data streams, we refer the interested reader to the extensive bibliographies in [5,22,24].

It is well known that algorithmic ideas developed in the context of parallel computational models have inspired the design of efficient algorithms in other models. For instance, Chiang et al. [8] showed that efficient external memory algorithms can be derived from PRAM algorithms using a general simulation. Aggarwal et al. [1] discussed how circuits with uniform linear width and polylog depth (NC) can be simulated efficiently in StrSort, providing a systematic way of constructing algorithms in this model for problems in NC that use a linear number of processors. Examples of problems in this class include undirected connectivity and maximal independent set.

Parallel techniques seem to play a crucial role in the design of efficient algorithms in the W-Stream model as well. For instance, the single-source shortest paths algorithm described in [10] is inspired by a framework introduced by Ullman and Yannakakis [29] for the parallel transitive closure problem. However, to the best of our knowledge, no general techniques for simulating parallel algorithms in the W-Stream model have been addressed so far in the literature.

1.1. Our contributions

In this paper, we show how classical parallel algorithms designed in the PRAM model can be turned into near-optimal algorithms in W-Stream for several classical combinatorial problems. We first show that any PRAM algorithm that runs in time T using N processors and memory M can be simulated in W-Stream using $p = O((T \cdot N \cdot \log M)/s)$ passes. This yields optimal tradeoff upper bounds of the form $p = O((n \cdot \text{polylog}\, n)/s)$ in W-Stream for several problems, where n is the input size. Relevant examples include list ranking and Euler tour of a tree. For other problems, however, this simulation does not provide good upper bounds. One prominent example concerns graph problems, for which efficient PRAM algorithms typically require $O(m + n)$ processors on graphs with n vertices and m edges. For those problems, this simulation method yields $p = O((m \cdot \text{polylog}\, n)/s)$ bounds, while $p = \Omega(n/s)$ almost-tight lower bounds in W-Stream are known for many of them [10].

To overcome this problem, we study an intermediate parallel model, which we call RPRAM, derived from the PRAM model by relaxing the assumption that a processor can only access a constant number of cells at each round. This way, we get the PRAM algorithms closer to streaming algorithms, since a memory cell in the working memory can be processed against an arbitrary number of cells in the stream. For some problems, this enhancement allows us to substantially reduce the number of processors while maintaining the same number of rounds. We show that simulating RPRAM algorithms in W-Stream leads to near-optimal algorithms (up to polylogarithmic factors) for several fundamental problems, including sorting, connected components, minimum spanning tree, biconnected components, and maximal independent set. We remark that, to the best of our knowledge, no previous algorithms were known for minimum spanning tree, biconnected components and maximal independent set in the W-Stream model.

A natural question is whether *ad hoc* techniques can yield better results than our general purpose simulation framework. We show that this is indeed the case for some problems, such as connected components, minimum spanning tree and biconnected components. However, as implied by our lower bounds, such improvements cannot be larger than polylog factors.

Finally, we show that there exist problems for which the increased computational power of the RPRAM model does not help in reducing the number of processors required by a PRAM algorithm while maintaining the same time bounds, and thus cannot lead to better W-Stream algorithms. An example is deciding whether a directed graph contains a cycle of length two.

1.2. Organization of the paper

The paper is organized as follows. Section 2 introduces our simulation techniques, that allow turning PRAM algorithms into efficient tradeoff W-Stream ones. Section 3 deals with the sorting problem, as a first application of our simulations.

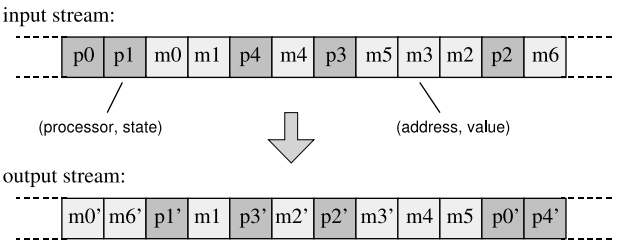

Fig. 1. Simulating a PRAM algorithm in W-Stream. Processor states and memory cells are updated as needed, and stored on the output stream in no particular order; unchanged memory cells are just propagated by copying them from input stream to output stream.

Section 4 focuses on graph problems: almost matching (up to polylog factors) upper and lower bounds are given for classical graph problems such as connected components, minimum spanning tree, biconnected components and maximal independent set. Section 5 outlines some limitations of our approach. Finally, Section 6 presents our conclusions, and some future research directions.

2. Simulating parallel algorithms in W-Stream

In this section we introduce general techniques for simulating parallel algorithms in W-Stream. We show in the next sections that our techniques yield near-optimal algorithms for many classical combinatorial problems in the W-Stream model. In Theorem 1 we discuss how to simulate Arbitrary CRCW PRAM algorithms. Throughout this paper, we assume that each memory address, cell value, and processor state can be stored using $O(\log M)$ bits, where M is the memory size of the parallel machine. In the Arbitrary CRCW PRAM model, both concurrent reads and concurrent writes are allowed; in the case of concurrent writes, an arbitrary processor succeeds.

Theorem 1. *Let A be an Arbitrary CRCW* PRAM *algorithm that uses N processors and runs in T parallel rounds using space $M = \text{poly}(N)$. Then A can be simulated in* W-Stream *in $p = O((T \cdot N \cdot \log M)/s)$ passes using s bits of working memory and intermediate streams of size $O(M + N)$.*

Proof. In the PRAM model, at each parallel round, every processor may read $O(1)$ memory cells, perform $O(1)$ instructions to update its internal state, and write $O(1)$ memory cells. A round of A can be simulated in W-Stream by performing $O((N \log M)/s)$ passes, where at each pass we simulate the execution of $\Theta(s/\log M)$ processors using s bits of working memory. The content of the memory cells accessed by the algorithm and the state of each processor are maintained on the intermediate streams in no particular order, as items of the form $(address, value)$ and $(processor, state)$, respectively (see Fig. 1). We simulate the task of each processor in a constant number of passes as follows. We first read from the input stream its state and the content of the $O(1)$ memory cells used by A and then we execute the $O(1)$ instructions performed. Finally, we write to the output stream the new state and possibly the values of the $O(1)$ output cells. Memory cells that remain unchanged are simply propagated through the intermediate streams by just copying them from the input stream to the output stream at each pass. □

The result of Theorem 1 can be generalized to work with Priority CRCW PRAM algorithms, as stated in the following corollary.

Corollary 1. *Let A be a Priority CRCW* PRAM *algorithm that uses N processors and runs in T parallel rounds using space $M = \text{poly}(N)$. Then A can be simulated in* W-Stream *in $p = O((T \cdot N \cdot \log M)/s)$ passes using s bits of working memory and intermediate streams of size $O(M + N)$.*

Proof. We follow the proof of Theorem 1. The main difference is that in the Priority CRCW PRAM model fixed priorities are assigned to processors, and if different processors write the same memory cell at the same round, the one with the highest priority prevails. This can be simulated in W-Stream by keeping track, for each cell, of whether it has been written at the current round, and if so of the priority of the processor that performed the write. A new write operation will be carried out only if the current processor has a higher priority. □

There are many examples of problems that can be solved efficiently in W-Stream using Theorem 1. For instance, solving list ranking in PRAM takes $O(\log n)$ rounds and $O(n/\log n)$ processors [3], where n is the length of the list. By Theorem 1, we obtain a W-Stream algorithm that runs in $O((n \log n)/s)$ passes. An Euler tour of a tree with n vertices is computed in parallel in $O(1)$ rounds using $O(n)$ processors [18], which by Theorem 1 yields again a $p = O((n \log n)/s)$ bound in W-Stream. We will now prove that both upper bounds are optimal in W-Stream.

First, we recall some results from [10], to which we will frequently refer in what follows. Consider the *element distinctness* problem, i.e., given a stream S of n numbers in $\{1, \ldots, n\}$, determine whether there are any duplicates in S. The following lower bound holds in W-Stream.

Theorem 2 (*[10]*)**.** *Solving element distinctness requires $\Omega(n/s)$ passes in* W-Stream*, where s is the space restriction in bits.*

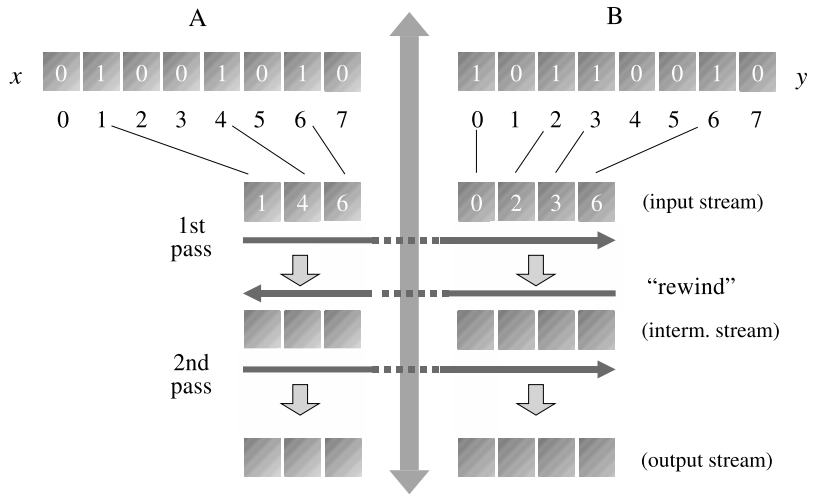

Fig. 2. Reduction from *bit-vector disjointness* to *element distinctness*, and distributed execution of a W-Stream algorithm for *element distinctness*: every time the boundary between A and B is crossed (dashed line) the content of the working memory (i.e., the algorithm's internal state) has to be transmitted.

Proof. The proof is based on a reduction from the *bit-vector disjointness* problem, i.e., two parties (A and B) have two n-bit vectors x and y, respectively, and want to know whether $x \cdot y > 0$. Bit-vector disjointness can be reduced to element distinctness in the following way: A creates a stream containing the indices corresponding to the 1's in her vector, and B does the same for his vector. Then they execute any W-Stream algorithm for element distinctness in a *distributed* fashion, i.e., at each pass, A runs the algorithm on her part of the stream, and then she sends the content of her working memory to B; upon receiving A's working memory, B resumes the execution of the algorithm on his part of the stream, and then sends the content of his working memory back to A (see Fig. 2). At the end of the execution, the streaming algorithm will determine whether all elements are distinct, thus also solving the bit-vector disjointness problem. Since any such execution requires the working memory image (algorithm's internal state) to be sent back and forth between A and B at each pass, and solving bit-vector disjointness requires transmitting $\Omega(n)$ bits [19], we obtain $p \cdot s = \Omega(n)$. □

By reducing bit-vector disjointness to undirected graph connectivity, the following theorem can also be proven.

Theorem 3 ([10])**.** *Solving undirected connectivity requires $\Omega(n/s)$ passes in* W-Stream*, where s is the space restriction in bits, and n is the number of nodes in the input graph.*

We will now prove that our algorithm for list ranking is optimal.

Theorem 4. *List ranking requires $\Omega((n \log n)/s)$ passes in* W-Stream*, where s is the space restriction in bits.*

Proof. The proof is based on a reduction from the problem of list ranking on an interconnection network [26]. Consider two distinct parties, Alice and Bob, each possessing some items of a list on which they want to solve the list ranking problem. As a worst case scenario, assume that each party has $n/2$ items and that all links go back and forth between them. In this case, as shown in [26], solving list ranking is at least as difficult as transferring an unknown number in $\{1, \ldots, (n/2)!\}$ between the parties, which in turn requires exchanging $\Omega(\log n!)$, i.e., $\Omega(n \log n)$ bits. Alice and Bob can reduce their problem to list ranking in the W-Stream model as follows. For each item in her portion of the list, Alice outputs an item on her stream (containing a list node unique id and the id of its successor), and Bob does the same for his items. Then they execute any W-Stream algorithm for list ranking in a *distributed* fashion (see Theorem 2). Since any such execution requires the working memory image (algorithm's internal state) to be sent back and forth between Alice and Bob at each pass, and solving list ranking on an interconnection network requires transmitting $\Omega(n \log n)$ bits, we obtain $p \cdot s = \Omega(n \log n)$. □

The same lower bound also holds for the computation of an Euler tour of a tree.

Corollary 2. *The computation of an Euler tour of a tree requires $\Omega((n \log n)/s)$ passes in* W-Stream*, for any space restriction of s bits.*

Proof. We observe that list ranking can be reduced to the computation of an Euler tour of a tree, as follows. The list induces a path, on which an Euler tour can be computed. It then suffices to walk the resulting tour beginning at the head of the list, to rank all its items. The result then follows from the lower bound for list ranking (see Theorem 4). □

There are problems, however, for which the bounds obtained through Theorem 1 are far from being optimal. For instance, efficient PRAM algorithms for graph problems typically require $O(m+n)$ processors, where n is the number of vertices, and m is the number of edges. For these problems, Theorem 1 yields bounds of the form $p = O((m \cdot \text{polylog}\, n)/s)$, while $p = \Omega(n/s)$ almost-tight lower bounds are known for many of them.

In Definition 1 we introduce RPRAM as an extension of the PRAM model. It allows every processor to handle in a parallel round not only $O(1)$ memory cells, but an arbitrary number of cells. Since in W-Stream a value in the working memory might be processed against all the data in the stream, we view RPRAM as a natural link between PRAM and W-Stream, even though it may be unrealistic in a practical setting.

Definition 1. An RPRAM (Relaxed PRAM) is an extended Priority CRCW PRAM machine with N processors and memory of size M where at each round each processor can execute $O(M)$ instructions that:

- can read an arbitrary number of memory cells. Each cell can only be read a constant number of times during the round, and no assumptions can be made as to the order in which values are given to the processor;
- can write an arbitrary subset of the memory cells. If different processors attempt to write the same cell at the same round, only the one with the highest priority is allowed to complete the operation. Writing can only be performed after all read operations have been done.

Similarly to a PRAM, each processor has a constant number of registers of size $O(\log M)$ bits. Differently from the Priority CRCW PRAM model, however, the priority of each processor can change at each round, and must be settled before performing any write operations.

The jump in computational power provided by RPRAM allows substantial improvements for many classical PRAM algorithms such as decreasing the number of parallel rounds while preserving the number of processors or reducing the number of processors used while maintaining the same number of parallel rounds. We show in Theorem 5 that parallel algorithms implemented in this more powerful model can be simulated in W-Stream within the same bounds of Theorem 1.

Theorem 5. *Let A be an* RPRAM *algorithm that uses N processors and runs in time T using space $M = \text{poly}(N)$. Then A can be simulated in* W-Stream *in $p = O((T \cdot N \cdot \log M)/s)$ passes using s bits of working memory and intermediate streams of size $O(M + N)$.*

Proof. We follow the proofs of Theorem 1 and Corollary 1. The main difference is that a processor in the RPRAM model can read and write an arbitrary number of memory cells at each round, executing many instructions while still using $O(\log M)$ bits to maintain its internal state. Since the instructions of algorithm A performed by a processor during a round do not assume any particular order for reading the memory cells, reading memory values from the input stream can still be simulated in one pass. Replacing cell values read from the input stream with the new values written on the output stream can be performed in one additional pass. □

3. Sorting

As a first simple application of the simulation techniques introduced in Section 2, we show how to derive efficient sorting algorithms in W-Stream. We first recall that n items can be sorted on a PRAM with $O(n)$ processors in $O(\log n)$ parallel rounds and $O(n \log n)$ comparisons [18]. By Theorem 1, this yields a W-Stream sorting algorithm that runs in $p = O((n \log^2 n)/s)$ passes. In RPRAM, however, sorting can be solved by $O(n)$ processors in constant time as follows. Each processor is assigned to an input item; in one parallel round it scans the entire memory and counts the numbers i and j of items smaller than and equal to the item the processor is assigned to respectively. Then each processor writes its own item into all the cells with indices between $i + 1$ and $i + 1 + j$, and thus we obtain a sorted sequence.

Theorem 6. *Sorting n items in* RPRAM *can be done in $O(1)$ parallel rounds using $O(n)$ processors.*

Using the simulation in Theorem 5, we obtain the result stated below.

Corollary 3. *Sorting n items in* W-Stream *can be performed in $O(n \log n/s)$ passes.*

We obtain a W-Stream sorting algorithm that takes $p = O((n \log n)/s)$ passes, thus matching the performance of the best known algorithm for sorting in a streaming setting [21]. Since sorting requires $p = \Omega(n/s)$ passes in W-Stream, as shown in Theorem 7, this bound is essentially optimal.

Theorem 7. *In* W-Stream*, sorting n items requires $p = \Omega(n/s)$ passes when the space restriction is s bits.*

Proof. It suffices to observe that the *element distinctness* problem (i.e., given a stream S of n numbers in $\{1, \ldots, n\}$, determine whether there are any duplicates in S) can be reduced to sorting plus one extra pass to detect duplicates in the sorted sequence. The result then follows from the $p = \Omega(n/s)$ communication complexity-based lower bound for element distinctness (see Theorem 2). □

We note, however, that both our algorithm and the algorithm in [21] perform $O(n^2)$ comparisons. We can reduce the number of comparisons to the optimal $O(n \log n)$ at the expense of increasing the number of passes to $O((n \log^2 n)/s)$ by simulating an optimal PRAM algorithm via Theorem 1, as stated before.

4. Graph problems

In this section we discuss how to derive efficient W-Stream algorithms for several graph problems using the RPRAM simulation in Theorem 5. Since efficient PRAM graph algorithms typically require $O(m + n)$ processors on graphs with n vertices and m edges [7], simulating such algorithms in W-Stream using Theorem 1 yields bounds of the form $p =$

$O((m \cdot \text{polylog } n)/s)$, while $p = \Omega(n/s)$ almost-tight lower bounds in W-Stream are known for many of them. Graph connectivity is one prominent example [10]. Notice that, assigning each vertex to a processor, RPRAM gives enough power for each vertex to scan its entire neighborhood in a single parallel round. Since many parallel graph algorithms can be implemented using repeated neighborhood scanning, in many cases this allows us to reduce the number of processors from $O(m + n)$ to $O(n)$ while maintaining the same running time. By Theorem 5, this yields improved bounds of the form $p = O((n \cdot \text{polylog } n)/s)$.

4.1. Connected components (CC)

The PRAM algorithm proposed by Shiloach and Vishkin [25] computes the connected components of a graph with n vertices and m edges in $O(\log n)$ parallel rounds, using $O(m + n)$ processors. We first describe the algorithm, and then we give an RPRAM implementation that uses only $O(n)$ processors while preserving the asymptotic number of parallel rounds, which, by Theorem 5, leads to a nearly optimal algorithm in W-Stream.

4.1.1. PRAM algorithm

During the execution of the algorithm, each vertex v is associated with a pointer field $D(v)$, which points either to another vertex or to v itself (initially $D(v) = v$ for all vertices). $(v, D(v))$ can be regarded as a directed edge in an auxiliary graph, called the *pointer graph*. While the pointer graph keeps changing during the execution of the algorithm, it is always a forest of *rooted trees* (a rooted tree is a directed graph in which (a) the underlying undirected graph is a tree, and (b) there exists a vertex r such that there is a directed path from each vertex to r) plus self-loops that occur only at the roots. As the algorithm proceeds, the number of trees decreases, while individual trees expand or disappear (this is caused by a *hooking* operation, through which one tree is attached to another). The trees are also subject to a *shortcut* operation that decreases their height. At the end of the algorithm, the vertices of each connected component will form a *rooted star* (i.e., a rooted tree of depth one) in the pointer graph. At the sth iteration the algorithm performs the following sequence of steps.

1. Each vertex updates its pointer as follows (*shortcut* operation):
   $$D_s(i) \leftarrow D_{s-1}(D_{s-1}(i))$$
   where $D_s(i)$ denotes the content of the pointer field of vertex i after the sth iteration;
2. For every vertex i that pointed to a root at the end of the previous iteration (i.e., if $D_s(i) = D_{s-1}(i)$), it is checked whether there is a neighbor j such that $D_s(j) < D_s(i)$ (it is assumed that each vertex is identified by a numeric label in $\{1, \ldots, n\}$). If such a vertex exists, then the root of i's tree is made to point to $D_s(j)$ (*hooking* operation), by executing:
   $$D_s(D_s(i)) \leftarrow D_s(j);$$
3. Each vertex i that points to a *stagnant* root (a root is stagnant at the sth iteration if its tree is stagnant, i.e., none of its vertices have been affected by any shortcut or hooking operation in the first two steps of s) checks whether any of its neighbors points to a vertex belonging to another tree. Upon finding such a vertex j (notice that it suffices to check whether $D_s(i) \neq D_s(j)$), i makes its root point to $D_s(j)$ (hooking operation). In order to determine whether a vertex v is stagnant, timestamp information is associated with it, recording when a new vertex was made to point to v (notice that if a root vertex is found to be stagnant, then it follows that its entire tree must be stagnant);
4. Each vertex i executes a second shortcut operation:
   $$D_s(i) \leftarrow D_s(D_s(i)).$$

The above steps are repeated until all trees are stagnant. The algorithm requires at most $\lfloor \log_{3/2} n \rfloor + 2$ iterations, using $n + 2m$ processors [25].

4.1.2. RPRAM implementation

We show how to implement each parallel round of the PRAM algorithm in $O(1)$ rounds in the RPRAM model, using only $O(n)$ processors. We notice that this algorithm does not rely on priorities, which will instead be crucial for the MST problem of Section 4.2. We attach a processor to each vertex. Steps 1 and 4 clearly require only $O(n)$ processors even in the PRAM model, as they perform a constant number of operations per vertex. Step 2 can be implemented using $O(n)$ processors as follows: first each vertex determines whether it is pointing to a root; if so, it scans its neighborhood looking for a vertex with smaller label; upon finding such a vertex it updates the pointer graph accordingly. Step 3 can be implemented as follows: first each vertex determines whether it is pointing to a stagnant root; if so, it scans its neighborhood, looking for a vertex pointing to another tree; upon finding such a vertex, it updates the stagnant root's pointer. We obtain the result in Theorem 8.

Theorem 8. *Solving CC in* RPRAM *takes $O(n)$ processors and $O(\log n)$ parallel rounds.*

By Theorem 5, this yields the following bound in W-Stream.

Corollary 4. *CC can be solved in* W-Stream *in $O((n \log^2 n)/s)$ passes.*

By the $p = \Omega(n/s)$ lower bound for undirected graph connectivity in W-Stream (see Theorem 3), this upper bound is optimal up to a polylogarithmic factor. This bound can be improved to $O((n \log n)/s)$ passes as shown in [10].

4.2. Minimum spanning tree (MST)

In this section, we first describe a PRAM algorithm for computing the MST of an undirected graph. We then give an RPRAM implementation that leads to an optimal algorithm (up to a polylog factor) in W-Stream by using the simulation in Theorem 5. Finally, we give an algorithm designed directly in W-Stream that slightly outperforms the algorithm obtained through the simulation.

4.2.1. PRAM algorithm

The CC algorithm introduced in Section 4.1.1 can be extended to find a minimum spanning tree in a (connected) graph, provided that we switch from the Arbitrary to the Priority CRCW PRAM model. It also takes $O(\log n)$ rounds and uses $O(m+n)$ processors. The algorithm is based on the property that given a subset V' of vertices, a minimum weight edge having one and only one endpoint in V' is in some MST. We assume that, at the beginning of the algorithm, processors are assigned to edges in such a way that the higher a processor's priority, the lighter the edge it is assigned to. Steps 2 and 3 of the CC algorithm undergo the following modification. Whenever a hooking operation is performed, the edge that caused it is flagged as belonging to the MST. Notice that, given the processor–edge association described above, if more than one hooking is possible between two trees, the minimum weight edge will always be preferred. This algorithm computes an MST in $O(\log n)$ parallel rounds.

4.2.2. RPRAM implementation

We assume that each edge is assigned a unique id, say in $\{1, \ldots, m\}$. We say that an edge has a priority inversely proportional to its weight (if two edges have the same weight, the edge with lowest id has higher priority). We attach a processor to each vertex. In order to implement steps 2 and 3 with $O(n)$ processors in $O(1)$ rounds, at the beginning of each round each processor scans its neighborhood, and assigns itself the priority of the highest priority edge incident to it, ignoring edges whose endpoints are in the same rooted tree. The implementation then proceeds as described in Section 4.1.2 for the CC algorithm. We obtain the result stated in Theorem 9.

Theorem 9. *MST is solvable in* RPRAM *using $O(n)$ processors and $O(\log n)$ parallel rounds.*

Assuming edge weights can be encoded using $O(\log n)$ bits, we obtain the following bound in W-Stream by Theorem 5.

Corollary 5. *MST can be solved in* W-Stream *in $O((n \log^2 n)/s)$ passes.*

We now give an algorithm designed directly in W-Stream that improves the bounds achieved by using the simulation.

4.2.3. A faster ad hoc W-Stream algorithm

We again assume edge weights can be encoded using $O(\log n)$ bits. We build the MST by progressively adding edges as follows. We compute for each vertex the minimum weight edge incident to it. This set of edges E' is added to the MST. We then compute the connected components induced by E' and contract the graph by considering each connected component as a single vertex. We repeat these steps until the graph contains a single vertex or there are no more edges to add. More precisely, we consider at each iteration a contracted graph where the vertices are the connected components of the partial MST so far computed. Denoting $G_i = (V_i, E_i)$ the graph before the ith iteration, the $(i+1)$th iteration consists of the following steps.

1. For each vertex $u \in V_i$, we compute a minimum weight edge (u, v) incident to u, and flag (u, v) as belonging to the MST (cycles that might occur due to weight ties are avoided by using a tie-breaking rule). Denote $E'_i = \{(u, v), u \in V_i\}$ the set of flagged edges.
2. We run a CC algorithm on the graph (V_i, E'_i). The resulted connected components are the vertices of V_{i+1}.
3. We replace each edge (u, v) by $(c(u), c(v))$, where $c(u)$ and $c(v)$ denote the labels of the connected components previously computed. If $c(u) = c(v)$ no edge is written.

We remark that the above algorithm is derived from a parallel version of Sollin's algorithm for MST, upon which several efficient PRAM algorithms have been based [4,9,30].

We now analyze the number of passes required in W-Stream. Let $|V_i| = n_i$. The first and the third steps require $O((n_i \log n)/s)$ passes each, since we can process in one pass $O(s/\log n)$ vertices. Computing the connected components also takes $O((n_i \log n)/s)$ passes, and therefore the ith iteration requires $O((n_i \log n)/s)$ passes. We note that at each iteration we add an edge for every vertex in V_i and thus $|V_{i+1}| \leq |V_i|/2$, i.e., the number of connected components is divided by at least two. We obtain that the total number of passes performed in the worst case is given by $T(n) = T(n/2) + O((n \log n)/s)$, which sums up to $O((n \log n)/s)$.

Theorem 10. *MST can be computed in $O((n \log n)/s)$ passes in* W-Stream.

By the $p = \Omega(n/s)$ lower bound for undirected graph connectivity in W-Stream (see Theorem 3), this upper bound is optimal up to a polylog factor.

4.3. Biconnected components (BCC)

Tarjan and Vishkin [28] describe a PRAM algorithm that computes the biconnected components (BCC) of an undirected graph in $O(\log n)$ time using $O(m+n)$ processors. We give an RPRAM implementation of their algorithm that uses only $O(n)$ processors while preserving the time bounds and thus can be turned using Theorem 5 in a W-Stream algorithm that runs in $O((n \log^2 n)/s)$ passes. We also give a direct implementation that uses only $O((n \log n)/s)$ passes.

4.3.1. PRAM algorithm

Given a graph G, the algorithm considers a graph G' such that vertices in G' correspond to edges in G and connected components in G' correspond to biconnected components in G. The algorithm first computes a rooted spanning tree T of G and then builds a subgraph G'' of G' having as vertices all the edges of T. The edges of G'' are chosen such that two vertices are in the same connected component of G'' if and only if the corresponding edges in G are in the same biconnected component. After computing the connected components of G'' the algorithm appends the remaining edges of G to their corresponding biconnected components. We now briefly sketch the five steps of the algorithm.

1. Build a rooted spanning tree T of G and compute for each vertex its preorder and postorder numbers together with the number of descendants. Also, label the vertices by their preorder numbers.
2. For each vertex u, compute two values, $low(u)$ and $high(u)$, as follows.

$$low(u) = \min(\{u\} \cup \{low(w) | p(w) = u\} \cup \{w | (u, w) \in G \setminus T\})$$
$$high(u) = \max(\{u\} \cup \{high(w) | p(w) = u\} \cup \{w | (u, w) \in G \setminus T\}),$$

   where $p(u)$ denotes the parent of vertex u.
3. Add edges to G'' according to the following two rules. For all edges $(w, v) \in G \setminus T$ with $v + desc(v) \leq w$, add $((p(v), v), (p(w), w))$ to G'', and for all $(v, w) \in T$ with $p(w) = v$, $v \neq 1$, add $((p(v), v), (v, w))$ to G'' if $low(w) < v$ or $high(w) \geq v + desc(v)$, where $desc(v)$ denotes the number of descendants of vertex v.
4. Compute the connected components of G''.
5. Add the remaining edges of G to their biconnected components. Each edge $(v, w) \in G \setminus T$, with $v < w$, is assigned to the biconnected component of $(p(w), w)$.

4.3.2. RPRAM implementation

We give RPRAM descriptions for all the five steps of the algorithm, each of them using $O(\log n)$ time and $O(n)$ processors. First, we compute a spanning tree of the graph using the RPRAM algorithm previously introduced. Rooting the tree and computing for each vertex the preorder and postorder numbers as well as the number of descendants are performed using list ranking and Euler tour [28], which take $O(\log n)$ time and $O(n)$ processors in PRAM, and thus in RPRAM. Since the second step takes $O(\log n)$ time using $O(n)$ processors in PRAM [28], the same bounds hold for RPRAM. We implement the third step in RPRAM in constant time and $O(n)$ processors, since it suffices a scan of the neighborhood for each vertex. For computing the connected components of G'' in the fourth step, we use the RPRAM algorithm previously introduced that takes $O(\log n)$ time with high probability and $O(n)$ processors. Finally, we implement the last step of the algorithm in RPRAM in $O(1)$ time and $O(n)$ processors by scanning the neighborhood for all vertices v and assigning the edges to the proper biconnected components. Since we implement all the steps of the algorithm in RPRAM in $O(\log n)$ rounds and $O(n)$ processors, we obtain the following result.

Theorem 11. *BCC can be solved in* RPRAM *using $O(n)$ processors in $O(\log n)$ parallel rounds.*

By Theorem 5, this yields the following bound in W-Stream.

Corollary 6. *BCC can be solved in* W-Stream *in $O((n \log^2 n)/s)$ passes.*

We now show that we can achieve better bounds with an implementation designed directly in W-Stream.

4.3.3. A faster ad hoc W-Stream algorithm

We describe how to implement directly in W-Stream the five steps of the parallel algorithm of Tarjan and Vishkin [28]. Notice that we have given constant time RPRAM descriptions for the third and the fifth step, thus by applying the simulation in Theorem 5 we obtain W-Stream algorithms that run in $O((n \log n)/s)$ passes. For computing the connected components in the fourth step, we use the algorithm in [10] that requires $O((n \log n)/s)$ passes. Therefore, to achieve a global bound of $O((n \log n)/s)$ passes, it suffices to give implementations that run in $O((n \log n)/s)$ passes for the first two steps. For the first step, we can compute a spanning tree within the bound of Theorem 10. Rooting the tree and computing the preorder and postorder numbers together with the number of descendants can be implemented in $O((n \log n)/s)$ passes using list ranking, Euler tour and sorting. Concerning the second step, we compute the low and $high$ values by processing $\Theta(s/\log n)$ vertices at each pass, according to the postorder numbers.

Theorem 12. *BCC can be solved in* W-Stream *in $O((n \log n)/s)$ passes.*

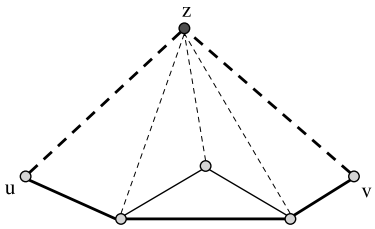

Fig. 3. Reducing *connectivity* to *biconnectivity*: every two vertices u and v are connected by two or more vertex-disjoint paths, if and only if the original graph is connected.

We will now show that both the upper bounds of Corollary 6 and Theorem 12 are optimal up to a polylogarithmic factor.

Theorem 13. *Let $G = (V, E)$ be an undirected graph, with n vertices and m edges. Solving BCC on G requires $p = \Omega(n/s)$ passes in* W-Stream*, for a space restriction of s bits.*

Proof. The proof is based on the well known observation that *biconnectivity* (i.e., the decision problem of whether any two vertices in a graph are biconnected) can be reduced to biconnected components. Graph connectivity in turn can be reduced to biconnectivity in the following way (see also Fig. 3). Add an extra vertex z and connect it to every vertex in V. The resulting graph G' is biconnected if and only if the original graph is connected (between any two vertices $u, v \in V$ there exist at least two vertex-disjoint paths: one in the original graph and the newly introduced path $\{(u, z), (z, v)\}$). The result then follows from the $p = \Omega(n/s)$ communication complexity-based lower bound for graph connectivity in W-Stream (see Theorem 3). □

4.4. Maximal independent set (MIS)

We give an efficient RPRAM algorithm for the maximal independent set problem (MIS), based on the PRAM algorithm proposed by Luby [20]. Using the simulation in Theorem 5, this leads to an efficient W-Stream implementation.

4.4.1. PRAM algorithm

A maximal independent set S of a graph G is incrementally built through a series of iterations, where each iteration consists of a sequence of three steps, as follows.

1. Compute a random subset I of the vertices in G, by including each vertex v with probability $1/(2 \cdot deg(v))$.
2. For each edge (u, v) in G, with $u, v \in I$, remove from I the vertex with the smallest degree.
3. Add to S the vertices in I, and then remove from G the vertices in I together with their neighbors.

The above steps are iterated until G gets empty. The algorithm uses $O(m + n)$ processors and $O(\log n)$ parallel rounds with high probability.

4.4.2. RPRAM implementation

We implement the first step of each iteration in constant time and $O(n)$ processors in RPRAM, since it requires each vertex to compute its own degree. The second step can also be implemented in constant time, by having each vertex in I scan its neighborhood, and remove itself upon encountering a neighbor also in I with a larger degree. Finally, we implement the third step in constant time as well by scanning the neighborhood of each vertex that is not in I, and removing it from G if at least one of its neighbors is in I. Since the algorithm performs $O(\log n)$ iterations with high probability [20], we obtain the bound in Theorem 14.

Theorem 14. *MIS can be solved in* RPRAM *using $O(n)$ processors in $O(\log n)$ rounds with high probability.*

By Theorem 5, this yields the following bound in W-Stream.

Corollary 7. *MIS can be solved in* W-Stream *in $O((n \log^2 n)/s)$ passes with high probability.*

We now show that the bound in Corollary 7 is optimal up to a polylog factor.

Theorem 15. *MIS requires $\Omega(n/s)$ passes in* W-Stream*, where s is the space restriction in bits.*

Proof. The proof is based on a reduction from the bit-vector disjointness communication complexity problem, defined as follows: Alice has an n-bit vector A and Bob has an n-bit vector B; they wish to know whether A and B are disjoint, i.e., $A \cdot B = 0$. To solve this problem, Alice and Bob build a graph on $4n$ vertices v_i^j, where $i = 1, \ldots, n$ and $j = 1, \ldots, 4$. If $A_i = 0$, then Alice adds to her stream edges (v_i^1, v_i^2) and (v_i^3, v_i^4), whereas if $B_i = 0$, then Bob adds to his stream edges (v_i^1, v_i^3) and (v_i^2, v_i^4). The size of any MIS is $2n$ if $A \cdot B = 0$ and strictly greater otherwise. Since solving bit-vector disjointness requires transmitting $\Omega(n)$ bits [19], and the distributed execution of any streaming algorithm requires the working memory image to be sent back and forth from Alice to Bob at each pass (see Theorem 2), we obtain $p \cdot s = \Omega(n)$. □

5. Limits of the RPRAM approach

In this section we prove that the increased power that RPRAM provides does not always help in reducing the number of processors from $O(m+n)$ to $O(n)$ in graph problems, and thus in obtaining W-Stream algorithms that run in $O((n \cdot \text{polylog } n)/s)$ passes. As an example, in Theorem 16 we prove that detecting cycles of length two in a directed graph takes $\Omega(m/s)$ passes.

Theorem 16. *Testing whether a directed graph with m edges contains a cycle of length two requires $p = \Omega(m/s)$ passes in* W-Stream.

Proof. We prove the lower bound by showing a reduction from the bit-vector disjointness two-party communication complexity problem. Assume that Alice has an m-bit vector A and Bob has an m-bit vector B. Alice creates a stream containing an edge $e(i) = (x_i, y_i)$ for each i such that $A[i] = 1$ and Bob creates a stream containing an edge $e'(i) = (y_i, x_i)$ for each i such that $B[i] = 1$, where $x_i = i \text{ div } \lceil\sqrt{m}\rceil$ and $y_i = i \bmod \lceil\sqrt{m}\rceil$. Let G be the directed graph induced by the union of the edges in the streams created by Alice and Bob. Clearly, there is a cycle of length two in G if and only if $A \cdot B > 0$. Since solving bit-vector disjointness requires transmitting $\Omega(m)$ bits [19], and the distributed execution of any streaming algorithm requires the working memory image to be sent back and forth from Alice to Bob at each pass (see Theorem 2), we obtain $p \cdot s = \Omega(m)$. □

Testing whether a directed graph has a cycle of length two can be easily done in one round by a PRAM using $O(m)$ processors, by just checking in parallel whether there is any edge (x, y) that also appears as (y, x) in the graph. This leads to an algorithm in W-Stream that runs in $O((m \log n)/s)$ passes by Theorem 5.

6. Concluding remarks

In this paper, we have introduced a general technique for simulating parallel algorithms in the W-Stream model. We have shown that this simulation yields optimal tradeoff algorithms for some problems, such as list ranking and Euler tour of a tree, while it leads to suboptimal results for others. Most notably, this is the case for graph problems, which in many cases require $O(n+m)$ processors to be solved efficiently, leading to W-Stream simulations that run in $O((m \cdot \text{polylog } n)/s)$ passes, while near-optimal direct W-Stream algorithms that run in $O((n \cdot \text{polylog } n)/s)$ passes are known for them (e.g., connected components).

To overcome this problem, we have introduced an intermediate model, called RPRAM, which is more powerful than the traditional PRAM model, and yet can be simulated in W-Stream within the same asymptotic bounds. Adapting PRAM algorithms to RPRAM usually allows us to decrease the number of processors from $O(n+m)$ to $O(n)$, yielding W-Stream algorithms for many graph problems (e.g., connected components, minimum spanning tree, biconnected components and maximal independent set) that are optimal up to a polylogarithmic factor.

Finally, we have shown that there are nonetheless graph problems for which the extra power provided by the RPRAM model does not help in reducing the number of processors from $O(m)$ to $O(n)$. One example is detecting whether a directed graph contains a cycle of length 2, for which a communication complexity-based lower bound of $p = \Omega(m/s)$ has been proven to exist.

Throughout this paper, we have assumed that only one input stream and one output stream can be accessed at any pass, and that the input data (e.g., edges of a graph) are given in arbitrary (adversarial) order. We believe that relaxing these constraints might open up some interesting research directions.

Aggarwal et al. [1] have shown that the ability to read and write two streams at any pass allows efficient solutions to problems, such as *alternating sequence*, that are hard even in StrSort. It would certainly be interesting to investigate the extra power provided by multiple (even $O(1)$) streams for more natural problems (e.g., fundamental graph problems).

Instead of being arranged in adversarial order, one might assume that the layout of the input stream is chosen uniformly at random from the set of all possible orderings (this scenario seems of interest as a form of average case analysis, or if each element of the input stream is an independent sample from some unknown distribution). Another possibility would be to limit the power the adversary has to reorder the input stream (random or otherwise). Guha et al. [16] have introduced the notion of *t-bounded adversary*, as an adversary with limited storage (so that he/she can delay at most t elements at a time), and have studied various selection problems in this context. It certainly seems worth analyzing also other classical combinatorial problems (including graph problems) in this model.

References

[1] G. Aggarwal, M. Datar, S. Rajagopalan, M. Ruhl, On the streaming model augmented with a sorting primitive, in: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science, FOCS'04, 2004.
[2] N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, Journal of Computer and System Sciences 58 (1) (1999) 137–147.
[3] R.J. Anderson, G.L. Miller, A simple randomized parallel algorithm for list-ranking, Information Processing Letters 33 (5) (1990) 269–273.
[4] B. Awerbuch, Y. Shiloach, New connectivity and MSF algorithms for shuffle-exchange network and PRAM, IEEE Transactions on Computers 36 (10) (1987) 1258–1263.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the 21st ACM Symposium on Principles of Database Systems, PODS'02, 2002, pp. 1–16.
[6] Z. Bar-Yossef, R. Kumar, D. Sivakumar, Reductions in streaming algorithms, with an application to counting triangles in graphs, in: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'02, San Francisco, California, 2002, pp. 623–632.
[7] G.E. Blelloch, B.M. Maggs, Parallel algorithms, in: The Computer Science and Engineering Handbook, 1997, pp. 277–315.
[8] Y. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vemgroff, J.S. Vitter, External-memory graph algorithms, in: Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'95, 1995, pp. 139–149.
[9] R. Cole, U. Vishkin, Approximate and exact parallel scheduling with applications to list, tree and graph problems, in: Proceedings of the 27th IEEE Symposium on Foundations of Computer Science, FOCS'86, 1986, pp. 478–491.
[10] C. Demetrescu, R. Finocchi, A. Ribichini, Trading off space for passes in graph streaming problems, ACM Transactions on Algorithms 6 (1) (2009).
[11] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, J. Zhang, On graph problems in a semi-streaming model, in: Proc. of the 31st International Colloquium on Automata, Languages and Programming, ICALP'04, 2004.
[12] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, J. Zhang, Graph distances in the streaming model: the value of space, in: Proceedings of the 16th ACM/SIAM Symposium on Discrete Algorithms, SODA'05, 2005, pp. 745–754.
[13] J. Feigenbaum, S. Kannan, M. Strauss, M. Viswanathan, An approximate L^1 difference algorithm for massive data streams, SIAM Journal on Computing 32 (1) (2002) 131–151.
[14] A.C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, M. Strauss, Fast, small-space algorithms for approximate histogram maintenance, in: Proceedings of the 34th ACM Symposium on Theory of Computing, STOC'02, 2002, pp. 389–398.
[15] L. Golab, M.T. Ozsu, Data stream management issues: a survey. Technical Report, School of Computer Science, University of Waterloo, TR CS-2003-08, 2003.
[16] S. Guha, A. McGregor, Approximate quantiles and the order of the stream, in: Proceedings of the 25th Symposium on Principles of Database Systems, PODS'06, 2006, pp. 273–279.
[17] M. Henzinger, P. Raghavan, S. Rajagopalan, Computing on data streams, in: External Memory Algorithms, in: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 50, 1999, pp. 107–118.
[18] J. Jájá, An Introduction to Parallel Algorithms, Addison-Wesley, 1992.
[19] E. Kushilevitz, N. Nisan, Communication Complexity, Cambridge University Press, 1997.
[20] M. Luby, A simple parallel algorithm for the maximal independent set problem, SIAM Journal of Computing 15 (4) (1986) 1036–1053.
[21] I. Munro, M. Paterson, Selection and sorting with limited storage, Theoretical Computer Science 12 (1980) 315–323.
[22] S. Muthukrishnan, Data streams: algorithms and applications. Technical Report. 2003. Available at http://athos.rutgers.edu/~muthu/stream-1-1.ps.
[23] J.M. Ruhl, Efficient Algorithms for New Computational Models. Ph.D. Thesis, Massachusetts Institute of Technology, September 2003.
[24] N. Schweikardt, Machine models and lower bounds for query processing, in: PODS'07: Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 2007, pp. 41–52.
[25] Y. Shiloach, U. Vishkin, An $O(\log n)$ parallel connectivity algorithm, Journal of Algorithms 31 (1) (1982) 57–67.
[26] J.F. Sibeyn, List-ranking on interconnection networks, Information and Computation 181 (2) (2003) 75–87.
[27] M. Sullivan, A. Heybey, Tribeca: a system for managing large databases of network traffic, in: Proceedings USENIX Annual Technical Conference, 1998.
[28] R.E. Tarjan, U. Vishkin, Finding biconnected components and computing tree functions in logarithmic parallel time, in: Proc. 25th Annual IEEE Symposium on Foundations of Computer Science, FOCS'84, 1984, pp. 12–20.
[29] J.D. Ullman, M. Yannakakis, High-probability parallel transitive-closure algorithms, SIAM Journal on Computing 20 (1) (1991) 100–125.
[30] C. Zaroliagis, Simple and work-efficient parallel algorithms for the minimum spanning tree problem, Parallel Processing Letters 7 (1) (1997) 25–37.